Code Smells

Data Class

Existem várias "*data classes"* ao longo do código do *JABREF*, como por exemplo a *UnknownExternalFileType* (*org.jabref.gui.externalfiletype*) ou a *AutoCompletionInput* (*org.jabref.gui.autocompleter*), que claramente não contém nenhum método que não seja de acesso a informação (*getters)*. Talvez este seja um mal necessário, mas quiçá haja métodos de manipulação de dados ou até mesmo noutras classes, que façam sentido para os encapsulados nesta classe. Estas possíveis alterações melhoram a qualidade do código, aumentando a sua organização e compressão, pois as operações de modificação de classes encontram-se todas no mesmo sítio.


UnknownExternalFileType.java


AutoCompletionInput.java

Switch Statements

Nas classes *CodeAreaKeyBindings* (*org.jabref.gui.keyboard*) e *TextInputKeyBindings* (*org.jabref.gui.keyboard*) existem dois *Switches* para os mesmos casos, isto é um indicador de que não estamos a aplicar corretamente os princípios de uma linguagem orientada aos objetos. Neste caso, talvez seja possível utilizar o *Template pattern* para que diferentes classes implementem cada caso. Esta alteração aumenta a organização do código e facilita futuras implementações e evita possíveis erros, pois não temos de alterar todos os *switches*, que estão localizados em classes diferentes.


CodeAreaKeyBindings.java


TextInputKeyBindings.java

Feature envy

Na classe *JabRefDesktop* (*org.jabref.gui.desktop*), o método *getNativeDesktop* acede maioritariamente à informação vinda da classe *OS* (*org.jabref.logic.util*), que por sinal serve para a deteção do sistema operativo, portanto o que faria mais sentido era mover este método para essa classe. Esta alteração melhora a legibilidade e arrumação do código.


JabRefDesktop.java


OS.java

Realizado por: Diogo Rosa 57464

Long method

In the logic package, inside the bst package there is a class called BibtexNameFormatter. This class has a method (formatName) in it which appears to be a long method.

In this code snippet we can see that the method is longer than it probably should be.

formatName.java

Data class

There is a class inside the logic.bibtext.comparator package called BibEntryDiff. This class is considered a data class since it only has data and its getters. Since this class is only used in one other class (ChangeScanner), a solution is to delete the data class and put its data and functionalities inside the ChangeScanner class.

Speculative generality

There is a class inside the logic.undo package called UndoRedoEvent. This is a subclass of the UndoChangeEvent, within the same class. However, this subclass does not have any data or functionalities other than its superclasses's. So, this class is being created perhaps with the intention to add other functionalities in the future but does not have them at the moment. A simple solution for this is to delete the subclass and use the superclass instead.

Realizado por: Tiago Fernandes 57677

Lazy Class

Existe um code smell na classe AutoSave *org.jabref.model.database.event.Autosave,* que identifiquei como uma lazy class. Esta classe não contém nem construtor, nem qualquer outro tipo de método, pelo que deve ser removida do jabref.

Long Method

Existe um code smell na classe Author *org.jabref.model.entry.Author,* que identifiquei com um long method. O método *addDotIfAbbreviation* contém inúmeras linhas de código, realizando demasiadas funcionalidades, que poderiam ser resolvidas em outros métodos. Por isso, como solução, deveriam ser criados outros métodos para reduzir esse mesmo método.

Feature Envy

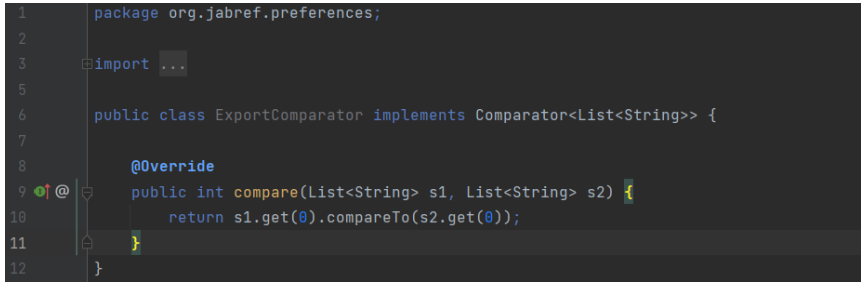
Este code smell encontra-se na classe BiblatexSoftwareEntryTypeDefinitions *org.jabref.model.entry.types.BiblatexSoftwareEntryTypeDefinitions,* que identifiquei como um feature envy. Esta classe utiliza mais métodos da classe BidEntryType do que os seus próprios métodos, pelo que uma solução possível era passar essas funcionalidades para a BidEntryType.

Realizado por: Tiago Duarte 58125

Lazy Class.

Every additional class adds more complexity to the project. More classes just mean more code to maintain. The ExportComparator class is not entirely too useful, has an empty constructor. In conclusion, this class does not do enough to earn the attention of the project. What is made in this class could be done in other classes.

A solution could be collapsing the class or possibly combining it with an existing class. Incline Class or Collapse Hierarchy can help clean up lazy classes if the single responsibility principle (a class should have only one reason to change) is being kept. The refactoring proposal could be implementing the compare method in the classes where it is most needed.



```java
package org.jabref.preferences;

import ...

public class ExportComparator implements Comparator<List<String>> {

    @Override
    public int compare(List<String> s1, List<String> s2) {
        return s1.get(0).compareTo(s2.get(0));
    }
}
```

Data Class.

Data class refers to a class which contains only fields and crude methods for accessing them (getters and setters). These are simply containers for data used by other classes. This class does not contain any additional functionality and cannot independently operate on the data that own. It’s a normal thing when a newly created class contains only a few public fields (and maybe even a handful of getters/setters). But the true power of objects is that they can contain behaviour types or operations on their data. In conclusion, this class which is a data class, contain only data and no real functionality, only a getter method. This indicates that it may not be a good abstraction or a necessary class. In this case that this class only uses public methods, one solution could be the use of the Encapsulation (which means, ability to conceal object data. Otherwise, all objects would be public and other objects could get and modify the data of the object without any checks and balances) to hide them from direct access and require that access to be performed via getters and setters only.

```java
public class VersionPreferences {

    private final Version ignoredVersion;

    public VersionPreferences(Version ignoredVersion) {
        this.ignoredVersion = ignoredVersion;
    }

    public Version getIgnoredVersion() {
        return ignoredVersion;
    }
}
```

Feature Envy.

This code smell occurs when there is a method that is more interested in the details of a class other than the one it is in. If two methods or classes are always talking to one another and seem as if they should be together, then chances are this is true. The getIgnoredVersion method reaches into the Version object to get the data on which it operates. It “wishes” that it were inside the Version class so that it could have direct access to the variable (ignoredVersion) it is manipulating.

In that case, we may consider moving this method (getIgnoredVersion) to the other class it uses (Version). This would make **classes more internally coherent** (because we would move a method to a class which contains all the data used by the method). So, in conclusion, the method would become a part of Version class instead of VersionPreferences class since all it does is to use a variable of the type of Version (ignoredVersion).

```
5   public class VersionPreferences {
6
7       private final Version ignoredVersion;
8
9       public VersionPreferences(Version ignoredVersion) {
10          this.ignoredVersion = ignoredVersion;
11      }
12
13      public Version getIgnoredVersion() {
14          return ignoredVersion;
15      }
16  }
```

Realizado por: Dinis Silvestre 58763

Long Parameter List

Located on line 275 of class Argument Processor, from path src/main/java/org/jabref/cli

Maximum of 7 parameters is advised for methods, this method has 9 parameters.

Refactoring: An auxiliary class containing the encoding(CharSet), filepreferences(FilePreferences), writeXMP(boolean) and embeddBibfile(boolean) and then having one object of that auxiliary class as a parameter would reduce the method's parameters to 6 and the same information could still be accessed. Doing this would solve the same code smell (long parameter list) for methods writeMetadatatoPDFsOfEntry(line 309), writeMetadatatoPdfByFileNames(line 343), writeMetadatatoPdfByCitekey(line 330), which also have those 4 parameters in common.

```
private void writeMetadatatoPdf(List<ParserResult> loaded, String filesAndCitekeys, Charset encoding,
                                XmpPreferences xmpPreferences, FilePreferences filePreferences,
                                BibDatabaseMode databaseMode, BibEntryTypesManager entryTypesManager,
                                FieldWriterPreferences fieldWriterPreferences,
                                boolean writeXMP, boolean embeddBibfile) {
```

Long Method

Located on line 181 of class Argument Processor, from path src/main/java/org/jabref/cli

Method with too many lines of code which makes it confusing and complex.

Refactoring: Some of the ifs could be on auxiliary methods (since there are many of conditions on this method), line 236 has a nested if which can be merged into a single if condition.

longmethod.txt

Message Chain

Code with structure similar to a typical message chain code smell, a.getB().getC().doSomething();

Refactoring: create instances before to minimize calls inside method.

```
exporter.get().export(databaseContext, Path.of(data[1]),
        databaseContext.getMetaData().getEncoding().orElse(preferencesService.getGeneralPreferences().getDefaultEncoding()),
        matches);
```

Realizado por: Artur Lourenço 63764

Design Patterns

Prototype

Podemos verificar este padrão no pacote org.jabref.logic.integrity, na classe *IntegrityMessage* onde a interface *prototype* é a *clonable* e a classe prototype concreta é a *IntegrityMessage,* esta classe é usada por diversas outras, também podemos verificar outras classes que implementam o *clonable*, como a *BibtexString* e a *BibEntry.* Este pattern permite-nos copiar objetos existentes sem fazer seu código ficar dependente de suas classes.

IntegrityMessage.java

Factory Pattern – ActionFactory

Podemos encontrar este *pattern* no package org.jabref.gui.actions onde a classe *ActionFactory* é a classe criadora, o produto pode ser considerado a interface *Action* (*org.jabref.gui.actions*) visto que esta é comum a todos os objetos criados e os outros parâmetros passados são classes java como o command, aqui a classe criadora fornece uma interface para criar objetos, mas permite que as se altere o tipo de objetos que serão criados, no caso podem ser criados menus e botões que podem realizar diversas ações.

ActionFactory.java

Action.java

Singleton Pattern – ExternalFileTypes

Podemos encontrar este *pattern* na classe *ExternalFileTypes* (*org.jabref.gui.externalfiletype*), onde o *singleton* é esta mesma classe e o cliente é por exemplo a classe *JabRefDesktop* (*org.jabref.gui.desktop*). Este *pattern* garante-nos que existe apenas uma instância da classe *ExternalFileTypes* ao longo de toda a execução do programa.

JabRefDesktop.java

ExternalFileTypes.java

Realizado por: Diogo Rosa 57464

Factory method

There is na abstract class in org.jabref.logic.exporter package called Exporter which is a normal class with some data and functionalities.

This Exporter class works as the product where the ExporterFactory (in the same package) works as the product's creator (factory). This factory class provides an interface which objective is to create Exporter objects, allowing its subclasses to change and deal with the object in a specific way.


Exporter.java


ExporterFactory.java

Composite

There is na interface in package org.jabref.logic.importer package called SearchBasedFetcher. This interface works as the component of the patterns and extends the WebFecther interface, located in the same package.

Working as the Leaf of this pattern we have the GrobidCitationFetcher in the package org.jabref.logic.importer.fetcher which implements the SearchBasedFecther interface, implementing both SearchBasedFecther and WebFetcher methods.
In the org.jabref.logic.importer.fetcher package we have a class called CompositeSearchBasedFetcher which implements the SearchBasedFetcher interface. Alike the Leaf class it also implements both SearchBasedFetcher and WebFetcher methods. This was identified as the composite class because it stores SearchBasedFetcher objects. This objects can be either leafs (GrobidCitationFecther) or even other composites (CompositeSearchBasedFetcher).


SearchBasedFetcher.java


WebFetcher.java


GrobidCitationFetcher.java


CompositeSearchBasedFetcher.java

Template method

There is an abstract class called BibDatabaseWriter in the org.jabref.logic.exporter package.

This class has a method that calls a template method. This template method is responsible for calling the multiple methods in the class (steps of the algorithm) that are implemented in the same class or by another class if the methods are abstract. The class in the JabRef tool that implements these abstract methods is the BibtexDatabaseWriter. This class extends BibDatabaseWriter and belongs to the same package as its superclass. This pattern is easily identified by looking at the template method in the abstract class, since it calls all steps that are implemented within the class or by its subclasses.


BibDatabaseWriter.java


saveDatabase.txt


savePartOfDatabase.txt


BibtexDatabaseWriter.java

Realizado por: Tiago Fernandes 57677

Singleton Pattern

Localização: src/main/java/org.jabref/logic/logging

A classe LogMessages possui um singleton pattern, consiste na existência de um construtor privado para seja possível controlar a quantidade de instâncias da classe LogMessages. Neste caso a classe guarda e aquiva todo o output de uma mensagem.

Factory Pattern

Localização: src/main/java/org.jabref/model/entry/field

A classe FieldFactory é a classe criadora que utiliza a classe OrFields para criar os seus objetos, já o produto é a interface Fields, que é a responsável por fornecer a interface para a criação dos objetos Fields.

Observer Pattern

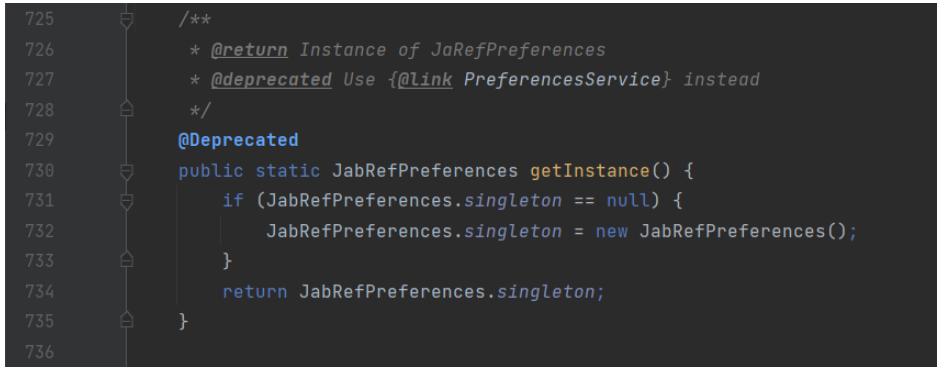
Localização: src/main/java/org.jabref/gui/fieldeditors

A classe LinkFilesEditor, utiliza a UiThreadObservableList, para poder observar objetos do tipo LinkedFileViewModel. Este pattern serve para que quando algo aconteça na classe observada, a que esteja a observar seja informada dessa mudança.

Realizado por: Tiago Duarte 58125

Singleton Pattern

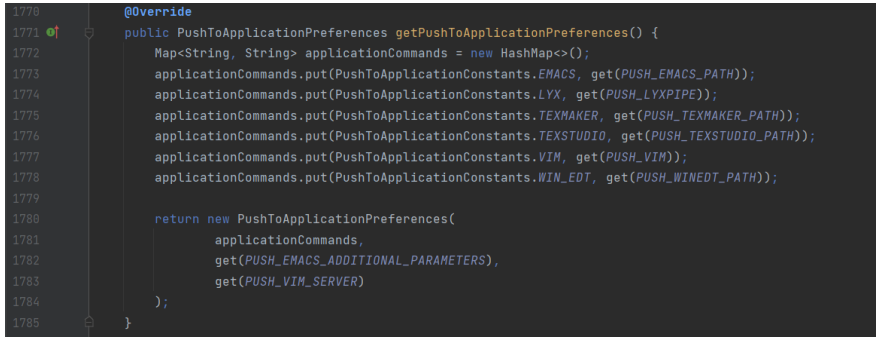
This type of design pattern comes under creational pattern as this pattern provides one of the best ways to create an object. This pattern involves a single class that is responsible to create an object while making sure that only a single object gets created. In this method, a new object is created. This class has only one instance and provides a global point to access it. This variable makes an object accessible, but it pollutes the namespace, and it still allows instantiating multiple objects of that class. So, the solution could be to make the class itself responsible for keeping track of its sole instance, which means, this class could ensure no other instance is created, by intercepting requests to create new objects or even the class could provide the sole way to access the instance



```
725     /**
726      * @return Instance of JaRefPreferences
727      * @deprecated Use {@link PreferencesService} instead
728      */
729     @Deprecated
730     public static JabRefPreferences getInstance() {
731         if (JabRefPreferences.singleton == null) {
732             JabRefPreferences.singleton = new JabRefPreferences();
733         }
734         return JabRefPreferences.singleton;
735     }
736
```

Command Pattern

In this case, the sender object can create a command object to encapsulate the order. The answer is the invoker object that invokes the command objects to complete whatever task it is supposed to do. The sender in this case is this method inside the class that creates a command (called applicationCommand) in which calls a method which is the receiver (get). The command pattern stores different requests (command objects - applicationCommand) into lists and manipulates them before they are completed. This pattern allows, as well, the command to be undone or redone. This pattern has various Benefits such as allowing commands to be manipulated as objects, its functionalities can be added to the command objects, such as putting them into queues and decoupling the objects of the software program.



```
1770    @Override
1771    public PushToApplicationPreferences getPushToApplicationPreferences() {
1772        Map<String, String> applicationCommands = new HashMap<>();
1773        applicationCommands.put(PushToApplicationConstants.EMACS, get(PUSH_EMACS_PATH));
1774        applicationCommands.put(PushToApplicationConstants.LYX, get(PUSH_LYXPIPE));
1775        applicationCommands.put(PushToApplicationConstants.TEXMAKER, get(PUSH_TEXMAKER_PATH));
1776        applicationCommands.put(PushToApplicationConstants.TEXSTUDIO, get(PUSH_TEXSTUDIO_PATH));
1777        applicationCommands.put(PushToApplicationConstants.VIM, get(PUSH_VIM));
1778        applicationCommands.put(PushToApplicationConstants.WIN_EDT, get(PUSH_WINEDT_PATH));
1779
1780        return new PushToApplicationPreferences(
1781                applicationCommands,
1782                get(PUSH_EMACS_ADDITIONAL_PARAMETERS),
1783                get(PUSH_VIM_SERVER)
1784        );
1785    }
```

Observer Pattern

The observer pattern is used when there is a one-to-many relationship between objects such as if one object is modified, its dependent objects are to be notified automatically. Observer pattern falls under the behavioural pattern category. The StateManager class uses the ObservableList (a list of tasks – which can be assigned to either isRunning or to getProgress), in order to observe objects of the type of the task, creating an array Observable which keeps the tasks in progress and the tasks running. The observer design pattern is also suitable for any scenario that requires push-based notification. The pattern defines a provider (also known as a subject or an observable) and zero, one, or more observers (depending on the length of the array). In short, observers register with the provider, and whenever the task state change occurs, the provider automatically notifies all observers by calling one of their methods.

```
57
58      private final ObservableList<Task<?>> backgroundTasks =
59
60              FXCollections.observableArrayList(task -> new Observable[]
61
62                      {task.progressProperty(), task.runningProperty()});
63
64
```

Realizado por: Dinis Silvestre 58763

Command

Located on path src/main/java/org/jabref/cli

It would be useful for when this class is called by class ArgumentProcessor (invoker) for us to store diferente commands so we can ‘redo’ them, since the same command may be rquested several times, or be held for later use. This also allows for a queue of commands to be executed in batch.

command.txt

Factory Method

Located on path src/main/java/org/jabref/gui/maintain

We can identify this design pattern since we have a factory class that controls the class instanciation, replacing constructors, and abstracting the process of object generation so that the object instantiated can be determined at run-time.

factory.txt

Composite

Located on path src/main/java/org/jabref/gui/sidepane

The class SidePaneComponent and SidePane demonstrate the typical Component design pattern

comp1.txt

comp2.txt

Realizado por: Artur Lourenço 63764

Metrics

Complexity Metrics

Complexidade ciclomática (*cyclomatic complexity*) é uma métrica de software que mede a quantidade de caminhos de execução independentes a partir de um código fonte. Esta métrica determina a estabilidade e confiabilidade do código, assim quanto menor este valor mais fácil é a interpretação e menor o risco de modificar um programa. Geralmente valores menores que 4 são considerados bons, entre 5 e 7 medianos e maiores ou iguais a 8 são valores demasiado elevados para a complexidade, assim, segundo o plug-in e analisando a vista do projeto conseguimos concluir que a média da complexidade ciclomática é 1,79 o que é um valor aceitável. Já na vista de pacotes podemos observar que o pior valor para a complexidade é 4,49 o que ainda é aceitável.

Já na vista de métodos conseguimos verificar quatro métricas diferentes: *cognitive complexity*, *essencial cyclomatic complexity, design complexity e a cyclomatic complexity.* A complexidade cognitiva mede o quão difícil uma certa unidade de código é de compreender intuitivamente e a complexidade de design como o nome indica mede o quão complexo é o design. Nesta vista conseguimos encontrar alguns valores excessivamente altos para estas métricas, isto pode ser resolvido, por exemplo para a complexidade cognitiva reduzindo o *nesting*, retirando o máximo de condições desnecessárias ou até mesmo entanto retirar *switches* (*Switch Statements code smell*), para a complexidade de design, reduzindo o tamanho dos métodos por exemplo (L*arge method code smell*) e para a complexidade ciclomática, tentar reduzir no geral o número de *code smells* que afetam a arrumação e legibilidade do código. Por sua vez a complexidade dos métodos vai também influenciar a complexidade das classes, juntamente com complexidade dos atributos e a resultado de herança e se resolvermos os problemas de complexidade para os métodos os valores encontrados nas classes vão melhorar ser muito mais aceitáveis.No geral, com algumas exceções, a complexidade ao longo de todas as vistas (métodos, classe, package, module e projeto) é aceitável e isto pode ser confirmado pela média do projeto como foi referenciado acima.

Realizado por: Diogo Rosa 57464

Line of code metrics

Introduction

Is this document I am going to point out some of the metrics of the project and then relate them with code smells.

The metrics were studied in 3 levels:

- Interface
- Class
- Method

Interface

There are 4 metrics results for each method:

- Comment lines of code (CLOC)
- Javadoc lines of code (JLOC)
- Lines of code (LOC)
- Non-comment lines of code (NCLOC)

We should analyze the CLOC. If we detect an interface with an excessive number of comments in the code, we may have a Comments code smell.

Class

There are 3 metrics results for each class:

- Comment lines of code (CLOC)
- Javadoc lines of code (JLOC)
- Lines of code (LOC)

The most important ones to analyze and relate to code smells are comment lines of code and lines of code.

A very high percentage of CLOC in total LOC could indicate that there might exist a Comments smell. Jabref has got a class with 824 CLOC and 879 LOC, which mean that approximately 94% of total LOC are CLOC. This does not mean that there is a code smell although it is a good indicator.

A very high number of LOC could also indicate that there might exist a Blob Class. Again, a high number of LOC in a class does not forcibly imply that there is in fact a smell.

Method:

There are 5 metrics results for each method:

- Comment lines of code (CLOC)
- Javadoc lines of code (JLOC)
- Lines of code (LOC)
- Non-comment lines of code (NCLOC)
- Relative lines of code (RLOC)

The important ones to analyze considering code smells are the CLOC, LOC and RLOC.

If we spot a method with a low CLOC/LOC ratio that might mean that we have a Comments smell. However, this ratio can mean nothing since these comments can be useful.

Methods with a high number of LOC might indicate that we have a long method smell. For example, we have a method with 324 lines which is pretty excessive for a method.

Tiago Fernandes 57677

Dependency Metrics

As Dependency Metrics são utilizadas para analisar as dependências que o projeto pode ter entre classes, interfaces e packages. Estas dependências são propicias à existência de code smells ou até mesmo a inconsistências no código. Se uma classe depende de várias classes, como por exemplo *JabRefFrame* com 115 dependências de outras, no caso de haver alguma mudança pode fazer com que algumas classes fiquem com inconsistências.

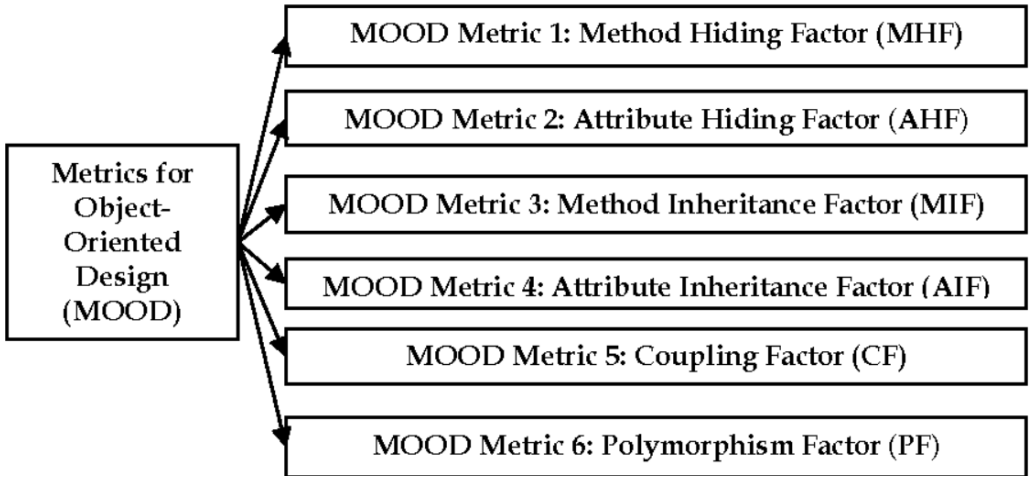
Os code smells que podem ficar associados às excessivas dependências de uma classe são: Shotgun Surgery, Feature Envy, Inappropriate Intimacy… O code smell Shotgun Surgery devido ao facto de poder ocorrer inconsistências e ser necessário mudar várias classes. Já o Feature Envy e o Inappropriate Intimacy poderão ocorrer devido a demasiadas chamadas de outras classes.

Realizado por: Tiago Duarte 58125

MOOD METRIC SET

- **Explanation of the collected metrics**

- The aim of this report is to produce a metrics-based overview of the codebase. To accomplish this, I selected metrics sets from those available in the MetricsReloaded plugin for IntelliJ IDEA. MetricsReloaded offers many metrics sets and I chose **MOOD Metrics**. The MOOD metric set is used to measure the properties of the system in which the design of the system is according to the concepts of Object-Oriented Design that is Encapsulation, Coupling, Inheritance, Information Hiding, and Polymorphism. The set contains six main metrics to measure the design of the system. The information which is analysed by this metric set are attribute hiding factor (AHF), attribute inheritance factor (AIF), coupling factor (CF), Method Hiding Factor (MHF), Method Inheritance Factor (MIF), and Polymorphism Factor (PM).



Metrics

Method Hiding Factor (MHF) and Attribute Hiding Factor (AHF)

To measure how the attributes and methods of one class are encapsulated, Method and Attribute hiding factors are being used. MHF and AHF show the average amount of how the members of a class are hidden in the system. MHF and AHF are 100% of all the members hidden, but in the real world, it is next to impossible because a class should communicate with other classes to provide different functionality of the system. It is not possible if the members are hidden.

To better understand how the percentage is made, let´s assume:

C = number of Classes

MV = number of other classes where a method is visible

AV = number of other classes where an attribute is visible

The MHF result (in percentage) is made as follows:

(Visible Methods) = (sum (MV) / (C–1)) / Number of Methods

MHF = 1 – (Visible Methods) → to get the hiding methods

The AHF result (in percentage) is made as follows:

(Visible Attributes) = (sum (MV) / (C–1)) / Number of Methods

AHF = 1 – (Visible Attributes) → to get the hiding attributes

If we have low MHF, then the methods are visible to almost every class. The methods are unprotected and can be subjected to changes by any class. The methods have a high tendency of reusability. In another case, if we have high MHF, then methods are not visible to many classes, and they cannot understand the working of it. These methods can be used only by the class containing it and it has less tendency of reusability. The ideal range should be around 8% to 25%. In this project, the MHF has a value of 36.93%, so it is way above the ideal range. In the case of AHF, the attributes of a class should be hidden from other classes so 100% is the ideal value of AHF. Generally, a high value of AHF is advised. In this project, the AHF has a percentage of 78.33%, it is not near 100%, but it is a high value.

Method Inheritance Factor (MIF) and Attribute Inheritance Factor (AIF)

In Inheritance, the child or subclass inherits the properties (Attribute and Methods) of the parent or superclass. The extent to which these methods and attributes are inherited is defined by Method Inheritance Factor (MIF) and Attribute Inheritance Factor (AIF).

To better understand how the percentage is made, let´s assume:

TM = total methods available in classes

TA = total attributes available in classes

The MIF result (in percentage) is made as follows:

MIF = inherited methods / TM

The AIF result (in percentage) is made as follows:

AIF = inherited attributes / TA

A child class that inherits many methods and attributes from its parent class contains a large value of MIF and AIF. As anything in a very small amount of very high amounts is harmful, the same case is with MIF and AIF. These values should be in a reasonable range. Generally, the range of MIF is between 20% to 80%. In this project, the MIF has a percentage of 18.28%, so a little under the ideal range. However, the ideal range of the AIF is between 0% and 48%. About this project, the AIF has a value of 23.20%, so inside the ideal range. The low value of AIF shows that the class should not inherit the attributes but rather the attributes should be private to the class. The child class can make some changes (for example, override) in the inherited methods.

Polymorphism Factor (PF)

To measure the degree or extent of method overriding by the child class from the parent or superclass, Polymorphism Factor (PF) is used. In polymorphism, the child class can implement the method in a different way. The same method can be implemented in different ways in the child and parent classes.

The PF result (in percentage) is made as follows:

PF = actual number of methods overrides / max number of total method overrides.

To keep the code clean and clear and to provide the high quality we can use high PF, but it increases the complexity of the system. In this project, the PF value is 49.59%. Polymorphism factor is integrally associated with method overriding.

Coupling Factor (CF)

If two or more clauses are related to each other by means of inheritance or aggregation or association then, in that case, these classes are said to be coupled. Many functionalities of the system can be done with the help of coupled classes. It is not advisable to have too many independent classes. The high value of CF shows that the classes of the system are more inter-connected and inter-dependent. This leads to the problem that sometimes it is very hard to change or repair the system in case of any bugs or problems because the functionality in which the bug is, could be implemented by more than two classes and we must make changes in all the related classes.

The CF result (in percentage) is made as follows:

CF = Actual coupling between different classes / maximum possible coupling that can happen in the system

If a class can access the method and attributes of the second class, then it is said that the first class is coupled with the second class. Generally, coupling happening due to inheritance is not considered while calculating the CF. The high value of coupling in the system leads to larger complexity. So, it is advisable to have a high value of cohesion and a low value of coupling. In this project, the CF has a value of 0.68%, which means the project has a good design architecture because contains less amount of coupling.

FACTOR	IDEAL MINIMUM (%)	IDEAL MAXIMUM (%)	MINIMUM TOLERANCE (%)	MAXIMUM TOLERANCE (%)	PROJECT (%)
MHF	12	22	9	36.9	36.93
AHF	75	100	67.7	100	78.33
MIF	66	78	61	84	18.28
AIF	53	66	37	75	23.20
PF	2	9	1	15	49.59
CF	0	11	0	24	0.68

With the help of the values, the conclusion is that the project is mostly out of the range accepted by the MOOD metrics set, which means the code and the way the project was made have problems, as detected by these metrics. According to the MOOD metrics, the overall quality of this objected-oriented project could be better.

How these metrics relate to the identified code smells

Code Metrics is a tool that analyzes our project, measures the complexity, and provides us better insight into the code. The advantages of Code Metrics are identifying code smells (it means identifying “the design flaws or bad practices, which might require attention. Some of the common code smells are Long Method, Duplicate Code, Large Class, and Dead Code), identifying the complexity and maintainability of our code (it will give an insight into our code maintainability and complexity) and increasing our Code Review efficiency.

Dinis Silvestre 58763

Martin Packaging Metrics

After running the codemetrics plugin utilizing the Marting Packaging metrics, I concluded that the JabRef project has very unstable packages, caused by a very low level of abstractness. We can verify an average value of 0,51 instability which is not a desirable value, since the ideal instability should show us values between 0 to 0.3 and 0.7 to 1, since packages should either be very stable (although this would mean they are harder to modify/update due to having greater responsabilities) or very unstable (which would mean there's a possibility of easy changes to these packages). In regards to the afferent coupling, which informs us of how many classes depend on a specific package (that should be ideally between 0 and 500) we can see that on average there's an afferent coupling of 316,87, which, on average, means we're good. The same can't be said about efferent coupling, that informs us of how many classes a specific package depends on. Efferent coupling should be in ranges of values between 0 and 20, since values higher than 20 means the package probably does more than it should and can therefore be a maintenance bottleneck and problematic to make changes in. We can see however, the average efferent coupling on project is 321,72, which is way higher that what we should aim for.

Of course all this is on average, and to find out what packages are really causing trouble on the project, we need to analyze the package metrics, and to find out code smells we're going to be looking into the more extreme values.

In regards to afferent coupling, we actually have several packages above our ideal range (0 to 500), but we'll focus on one of the packages with extreme values. *org.jabref.gui* is one example of this, with an afferent coupling of 2253. This means 2253 classes depend on this package. The fact it has several data classes and feature envy code smells (as identified by one of my colleagues) are evidences that there are definitely dependencies here that could be reduced by refactoring some code smells.

About efferent coupling, I'll analyze the package *logic.bibtex.comparator* which has an efferent coupling value of 822, meaning it depends on 822 classes. The data class identified by one of my colleagues on this package shows us how refactoring some code smells and getting rid of some unnecessary classes here would also reduce the coupling.

In conclusion, there are many more concerning values on these metrics (one package with efferent coupling of 5580, and another one with 13210 of afferent coupling!) which would surely benefit from having code smells identified and refactored making the JabRef project alot easier to maintain.

Artur Lourenço 63764

Use Case Diagram descriptions

ID: 1

No *use case diagram* da aba *edit*, considerei que apenas existia um ator, o utilizador do *jabref*, como todas os casos nesta aba apenas lidam com funcionalidades e objetos dentro do *jabref* penso que não haja mais atores.

Caso Rank

O caso *rank*, têm seis <<*extends*>>, que basicamente são variações do *rank*, temos os casos *rank1, rank2*, … e rank5, que não passa de uma avaliação de 1 a 5 e temos o caso *Clear Status* que limpa esta avaliação.

Caso Priority

O caso *priority*, têm quatro <<*extends*>>, que basicamente são variações do *priority*, temos os casos *set priority to low, set priority to medium* e *set priority to high* que definem os diferentes níveis de prioridade e temos o caso *Clear Priority* que limpa esta avaliação.

Caso Read Status

O caso *Read Status*, têm três <<*extends*>>, que basicamente são variações do *Read Status*, temos os casos *set read status to read, set read status to skimmed* que definem os se o documento foi ou não foi lido e por fim temos o caso *Clear Read Status* que limpa esta avaliação.

Caso cut/copy

No caso do cut existe um <<*include*>> que aponta para o copy, pois um cut não passa de um *copy* seguido de uma eliminação do objeto selecionado.

O resto são casos isolados que não têm ligações com os demais.

Realizado por: Diogo Rosa 57464

ID: 2

This use case diagram represents what happens in a library existing.

The library offers some funcitionalities that the user can use:

- Add Entry
- Click Entry
- Edit Entry
- Delete Entry
- Change properties
- Add Entry from plain text
- Edit preamble
- Edit String constrants
- Edit citation key patterns

There is only 1 primary actor (Client) using the library because he decides what actions to take in the library.

There is also only 1 secondary actor (Web) who reacts to the requests made by the primary actor (Client).

Use cases:

- Add Entry
- Web Search
- Create Entry
- Preview entries
- Fail Search
- Download entry
- Click Entry
- Edit Entry
- Delete Entry
- Change properties
- Add Entry from plain text
- Edit preamble
- Edit String constrants
- Edit citation key patterns

Add entry:

This is the most complex use case since there is 2 ways the client can add an entry: through Web Search or creating a new entry (Web Search and Create entry generalizes to Add Entry).

If we decide to add an already existing entry through the Web we need to make a web search. When we make this search the Web gives us a preview of the entries we might want to add (Web Search includes Preview entries). If there is no entries, the search fails and no entries appear on the preview (Fail search extends Preview entries). The client then decides if he wants to download an entry showed in the preview in order to add it to the library (Download entry extends Preview entries).

Click entry:

The user can click an entry in the library in order to view it. The client can also edit the entry when it's clicked. Therefore, the Edit entry use case extends Click entry.

Realizado por: Tiago Fernandes 57677

ID: 3

Este diagrama de use case representa a aba file do JabRef que por sua vez contém inúmeras funcionalidades.

Este diagrama apenas contém um ator primário cliente que é a representação do utilizador da aplicação JabRef.

Casos do File:

- Add Group
- Add Library
- Close
- Recent Libraries
- Open Library
- Import
- Export
- Connect to Shared Database
- Save
- Quit
- Discard Changes

Add Group

O grupo apenas pode ser adicionado caso uma biblioteca tenha sido adicionada e caso o cliente queira.

Add Library

A biblioteca criada na aplicação JabRef para futuramente poder receber novas entradas.

Close

Fecha a biblioteca selecionada, por isso apenas poderá existir caso haja biblioteca.

Recent Libraries

Abre a última biblioteca a ser fechada. Apenas pode ser chamado caso haja biblioteca aberta. Interage com o ator secundário Local que funciona com base de dados do teu dispositivo.

Open Library

Abre uma biblioteca localizada no teu dispositivo. Interage com o ator secundário Local que funciona com base de dados do teu dispositivo.

Import

Importa uma biblioteca localizada no teu dispositivo. Interage com o ator secundário Local que funciona com base de dados do teu dispositivo.

Export

Exporta uma biblioteca para o teu dispositivo. Interage com o ator secundário Local que funciona com base de dados do teu dispositivo.

Connect to Shared Database

Conecta uma base de dados de um servidor especificado pelo cliente. Interage com o ator secundário Server, que fornece a base de dados desejada.

Save

Guarda as bibliotecas no dispositivo. Interage com o ator secundário Local que funciona com base de dados do teu dispositivo.

Quit

Sai da aplicação. Caso ainda não tenha guardado tem a opção de guardar, ou descartar as alterações (Discard Changes).

Realizado por: Tiago Duarte 58125

ID: 4

A use case is a written description of how users will perform tasks on your website. It outlines, from a user's point of view, a system's behaviour as it responds to a request. Each use case is represented as a sequence of simple steps, beginning with a user's goal, and ending when that goal is fulfilled.

This use case diagram represents what happens in the JabRef program. This program offers some functionalities that the user can use:

- Access File
- Access Edit
- Access Library
- Access Quality
- Access Lookup
- Access Tools
- Access View
- Access Options
- Access Help

The primary actor of this specific use case diagram is the client because it is the only stakeholder that calls on the system to deliver one of its services and it is the only one who decides what actions to take in the JabRef program. The client is, in this case, the actor who triggers the use case. There are also 4 secondary actors (Server – one specific server, Local – the local database, Web – global database, and Admin – admin of the program) who react to the requests made by the primary actor (Client).

Each of my teammates made a different use case, specifying File, Edit, Library and Quality. The JabRef Use Case has the duty of specifying the big picture – the program itself.

Functionalities

The Access File functionality communicates with the server and with the local database when requests, for example, *Saved Library – for Local (in order to connect to the local database) and Connect to Shared Database - for Server* (in order to connect to a database of a specific client specified by the client), and that is why this functionality has these two second actors.

The Access Edit functionality does not communicate with any secondary actor, because of all the options the user can choose none of them has the need to connect to either a server, local, web or admin.

The Access Library functionality communicates with the web when requests, for example, the *New Entry* - there are 2 ways the client can add an entry: through Web Search or create a new entry. If we decide to add an already existing entry through the Web, we need to make a web search. When we make this search the Web gives us a preview of the entries we might want to add. And that is why this functionality has this second actor.

The Access Quality functionality communicates with the local and with the web when requests (for example, *Find Duplicates* – for local and *Abbreviate Journal names* – for Web) and that is why this functionality has these two second actors.

The Access Lookup functionality communicates with the web when requests (for example, *Search Document Identifier Online* – requests a web search) and that is why this functionality has only one second actor.

The Access Tools functionality communicates with the local when requests (for example, *Search for citations in LaTex files* – requests a local search) and that is why this functionality has only one second actor.

The Access View functionality does not communicate with any secondary actor, because of all the options the user can choose none of them has the need to connect to either a server, local, web or admin.

The Access Options functionality does not communicate with any secondary actor, because of all the options the user can choose none of them has the need to connect to either a server, local, web or admin.

The Access Help functionality communicates with the admin when requests (for example, *Online Help* – requests an admin help) and that is why this functionality has only one second actor.

ID: 5

Description: The quality use case was based on the “quality” tab on JabRef and it’s options. For the most part, the use cases only have one actor since the client selects a certain option and that option happens inside the system without involving any secundary actor. The only exception happens on the use cases related to the journals (abbreviating and unabbreviating journal names) since that will change the journal entries on the database on the database, which can either be the local repository or the online repository. When the primary actor (metioned only as “client” from now on) clicks on “Find duplicates”, what happens is the system (JabRef) looks for duplicate entries. On the “merge entries” use case, what happens is the system merges two entries if two entries are selected. On “Check integrity” the system looks for problems on the entries. On “cleanup entries” the system cleans up the entries according to a set of several options that shows up when the button is clicked. “Automatically set file links” set external links. “Abbreviate journal names” abbreviates the journal’s names according to a specific specification, such as “DEFAULT”, “MEDLINE” or “SHORTEST UNIQUE”. “Unabbreviate journal names” sets the journal’s names to their unabbreviated names, undoing the abbreviation if such happened.

Primary Actors: Client – The User of JabRef.

Secondary Actors: Database – Can be the local or online repository.